Le Monkey Patching

Michaël Spawn - Mai 2012

Résumé

Dans cet article, je vais tâcher de présenter une pratique de programmation douteuse appelée le « Monkey Patching », ou la « modification du singe » (aussi appelée le « Guérilla Patching ». Ainsi que de développer son intérêt dans la rédaction de scripts pour le logiciel *RPG Maker*.
Sachez tout d'abord que je ne suis pas un grand adepte de ce genre de pratique, mais que dans les exemples présentés dans ce papier, le Monkey Patching me semble être une solution intéressante.

Introduction

Le terme Monkey Patching désigne la modification/l'extension de code source sans altérer la source originale (principalement dans les langages de programmations dynamiques). On peut considérer ça comme une extension de classe déjà déclarée (dans le cadre d'un langage orienté objets).
Concrètement, le Monkey Patching me permet d'ajouter/de greffer, de modifier des éléments d'une classe sans modifier le code original.
Par exemple, je déclare une classe *Voiture* et un peu après, je lui ajoute une méthode *rouler*. Un autre exemple serait d'ajouter une méthode *getFirst* à ma classe *String* qui me retournerait le premier caractère d'une chaîne de texte.
Certains langages permettent ce genre de pratique, je citerai non-exhaustivement *Ruby*, *JavaScript* ou encore *Python*.
Cet article portera principalement sur *Ruby*.

Opinion sur le Monkey Patching

Bien que cela puisse paraître assez alléchant, le Monkey Patching pose énormément de problème de « raisonnement », en effet, nous sommes généralement habitués à lire notre code de manière linéaire, en abusant du Monkey Patching, lire un code (et donc, par extension, le raisonner), devient beaucoup plus complexe car nous ne savons jamais si le code que nous lisons est la « version finale ». Peut être qu'un patch a été rédigé plus loin, altérant un comportement défini.
Donc, dans un projet classique, l'abus de ce genre de pratique peut entraîner une perte de temps conséquente dans la relecture du code.

Exception, les potentiels bienfaits du Monkey Patching

Dans certains cas, le Monkey Patching peut être une bénédiction. Comme exemple, je vais citer les logiciels *RPG Maker* édités par *Enterbrain* qui offrent (depuis leur version *XP*) une solution de programmation embarquée (utilisant le langage *Ruby*).

Un outil destiné aux non-programmeurs

RPG Maker est un jeu pour faire des jeux, dans ce sens, il n'est pas demandé à l'utilisateur de maîtriser le langage *Ruby* pou réaliser son jeu. La programmation n'est qu'un atout complémentaire pour pousser la personnalisation un peu plus loin. C'est donc naturellement qu'une nouvelle classe d'utilisateurs est née, les programmeurs, qui sans pour autant se lancer dans la création de jeux complets, produisent des patchs pour le logiciel en lui greffant/ en modifiant de nouvelles composantes.

Sans le Monkey Patching, l'ajout de ces patchs auraient demandés à l'utilisateur, une connaissance de *Ruby* (assez sommaire cela dit) et de modifier le code original du jeu. Grâce au Monkey Patching, il est possible d'ajouter ces patchs les uns à la suite des autres assurant une certaine compatibilité (dans la mesure du possible).

Pourquoi Monkey Patcher plutôt que modifier le code original

Voici une petite liste des intérêts du Monkey Patching (dans le cadre de *RPG Maker)* non-exhaustive.

- Possibilité de revenir au code d'origine facilement
- Au moyen d'alias, assurer une compatibilité sensible
- Réduire le nombre de modifications
- Dans le cas de la lecture d'un script, comprendre ses modifications.

Je pense tout de même que le principal intérêt est de pouvoir facilement restaurer le code d'origine et de maximiser les inter-compatibilités.

Les alias, une des armes du Monkey Patching

Nous avons vu que nous pouvions « écraser » des méthodes, des attributs, mais les alias nous permettent de changer le nom d'une méthode. Cela peut être très pratique dans le cas ou je dois, par exemple, modifier une méthode pour lui greffer des actions. Je n'ai qu'a créer un alias de la méthode, réécrire la signature de la méthode, dans ma nouvelle méthode j'appelle l'alias et à la suite, j'écris mes

modifications.
Malgré le côté très naïf de la chose, c'est un maximisant les alias faisables qu'on peut augmenter la compatibilité. Par exemple, admettons que j'écrive un patch pour ajouter à ma classe *Game_Party* un compteur de combat remportés. La solution « triviale » serait de modifier directement la classe. Mais admettons qu'un autre patch sorte pour ajouter une gestion des quêtes, si les 2 programmeurs n'utilisent pas les alias, les modifications d'un patch écraseront les modifications de l'autre. Alors que si l'un alias le constructeur de *Game_Party,* l'appelle dans son nouveau constructeur puis ajoute sa modification et que le suivant alias (une fois de plus) le constructeur de *Game_Party* l'appelle, et fait ses modifications, les 2 seront prises en comptes.

Conclusion

Je n'ai volontairement pas mis d'exemple de code car il s'agit plus d'exemples théoriques. Je maintiens que le Monkey Patching rend la lecture du code compliqué, mais dans le cas de patchs pour *RPG Maker* (par exemple) ils peuvent garantir une bonne compatibilité et une possibilité de revenir en arrière facilement !

Liens complémentaires (et probables sources)

- http://en.wikipedia.org/wiki/Monkey_patch
- http://www.ruby-doc.org/docs/ProgrammingRuby/
- http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_classes.html
- http://funkywork.blogspot.com/2011/12/metaprogrammation-et-ruby.html
- http://ruby.about.com/od/rubyfeatures/a/aliasing.htm